Implicitly-Defined Neural Networks for Sequence Labeling

Michaeel Kazi
MIT Lincoln Laboratory
244 Wood St, Lexington, MA, 02420, USA
michaeel.kazi@ll.mit.edu

Abstract

We relax the causality assumption in formulating recurrent neural networks, so that the hidden states of the network are all coupled together. This goes beyond bidirectional RNN, which consists of two explicit recurrent networks concatenated together. The motivation behind doing this is to improve performance on long-range dependencies, and to improve stability (solution drift) in NLP tasks. We choose an implicit neural network architecture, show that it can be computed reasonably efficiently, and demonstrate it proof-of-concept on the task of part-of-speech tagging.

1 Introduction

Feedforward neural networks were designed to approximate and interpolate functions. Recurrent networks were developed to predict sequences. These recurrent networks can be 'unwrapped', and thought of as a very deep feedforward network, with each layer sharing the same set of weights. Computation proceeds one step at a time, like the trajectory of an ordinary differential equation when solving an initial value problem. However, in certain applications in natural language processing, especially those with long-distance effects, and where grammar matters, sequence prediction may be better thought of as a boundary value problem. In this work, we propose challenging the traditional left-to-right causality of neural networks, and demonstrate the feasibility and potential power of implicit methods.

2 Comparison with previous work

Long-range dependencies have been an issue as long as there have been NLP tasks, and there are many effective approaches to dealing with them. In the context of HMMs, there are the "Forward-Backward" models. In information extraction, there are non-local sequence models that use Gibbs sampling (Finkel et al., 2005). In recent years, there is the bidirectional LSTM (Schuster and Paliwal, 1997) that incorporates past and future hidden states via two separate recurrent networks. Unlike most of the previous methods, this method is not able to be simplified into a dynamic programming technique. The resulting states for each sequence element are more strongly coupled to each other, with potential for emergent effects. Solving the equations is more difficult, however, and make use of techniques for solving nonlinear systems of equations.

3 Recurrent Neural Networks

A typical recurrent neural network has the following specification:

- Input sequence $[x_1, x_2, \ldots, x_s]$
- Initial state given as order k tensor h_0
- Transition Function $h' = f(x, h)$

Produce order $k + 1$ tensor:

$$[h_0, h_1 = f(x_1, h_0), h_2 = f(x_2, h_1), \ldots, h_s = f(x_s, h_{s-1})]$$

Long-short-term-memory (Hochreiter and Schmidhuber, 1997), gated-recurrent (Cho et al., 2014), and its variants follow this formula, with different choices for the state transition function. Computation proceeds left-to-right, with each next state depending on the previously computed hidden state. It is a very reasonable assumption, and it makes computation straightforward and tractable. However, what would happen without it? Suppose

This work is sponsored by the Air Force Research Laboratory under Air Force contract FA-8721-05-C-0002. Opinions, interpretations, conclusions and recommendations are those of the authors and are not necessarily endorsed by the United States Government.

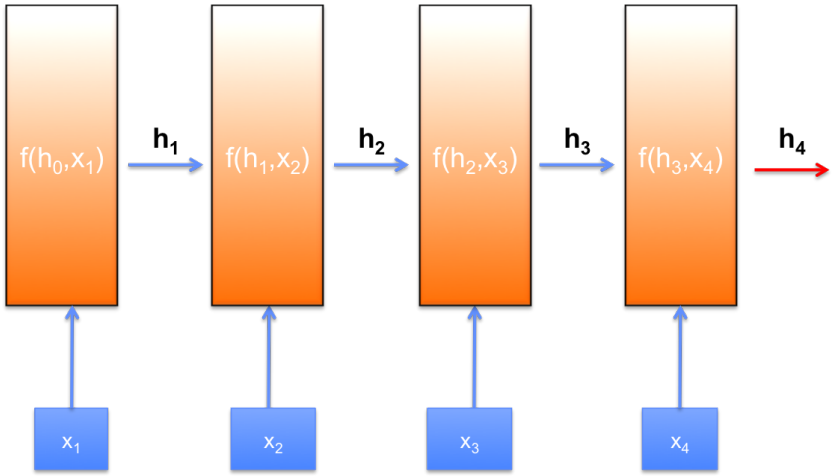

Figure 1: Traditional recurrent neural network structure.

$h_t = f(x_t, h_{t-1}, h_{t+1})$, or an even wider stencil? We arrive at a system of nonlinear equations. This setup has the potential for arriving at nonlocal, whole-sequence dependent results. Analogously to the theory of ODEs, it may also be more 'stable', whereby the predicted sequence may drift less from the true meaning, since errors will not compound with each time step in the same way.

4 Implicit RNN

4.1 Architecture for this work

There are many possibilities of how to architect a neural network – in fact, this is one of its best features – but we restrict our discussion to the one depicted in Figure 2. In this setup, we have the following variables:

input data	X
input labels	Y
parameters	θ
transformed input	ξ
hidden layers	H

and functions:

loss function	$L = \ell(\theta, H, Y)$
implicit hidden layer definition	$H - F(\theta, \xi, H) = 0$
input layer transformation	$\xi = g(\theta, X)$

Our implicit definition function, F, is made up of local state transitions, and forms a system of nonlinear equations that we need to solve:

$$\begin{aligned} h_1 &= f(h_0, h_2, \xi_1) \\ &\dots \\ h_i &= f(h_{i-1}, h_{i+1}, \xi_i) \\ &\dots \\ h_n &= f(h_{n-1}, h_{n+1}, \xi_n) \end{aligned}$$

4.2 Computing the forward pass

To evaluate the network, we must solve the equation $H = F(H)$. We computed this via an approximate Newton solve:

$$\begin{aligned} H_{n+1} &= H_n - (I - \nabla_H F)^{-1}(H_n - F(H_n)) \\ H_{n+1} &\approx H_n - P(\nabla_H F)(H_n - F(H_n)) \end{aligned}$$

We use the geometric series polynomial $P_n(x) = 1 + x + x^2 + \ldots + x^n$, which converges provided that $||\nabla_H F|| < 1$. For most of our experiments we even let $n = 1$. Note that $n = 0$ is fixed point iteration, which has also worked in experiments, albeit with slower convergence.

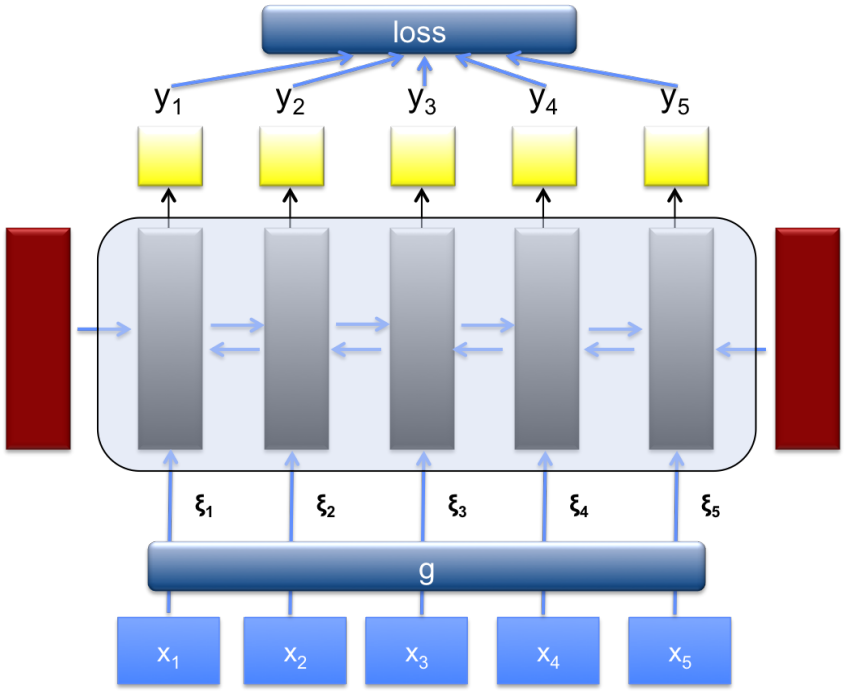

Figure 2: For the remainder of this paper, we will focus on the following 'word classification' architecture.

The assumption that $||\nabla_H F|| < 1$ probably works due to the initialization of the network. Weight matrices for the bidirectional transition function (see Section 4.4) strongly affect the size of the overall gradient, and we initialize all parameters with the range $[-.1, .1]$. In experiments, as the magnitudes of the weights grow, the solver takes more iterations, however, it does not seem to reach a point where the approximation is wholly inadequate.

4.3 Gradients

In order to train the model, we perform gradient descent. Taking the gradient of the loss function:

$$\nabla_\theta L = \nabla_\theta \ell + \nabla_H \ell \nabla_\theta H$$

so we will need to know the gradient of the hidden units with respect to the parameters, which we can find via the implicit definition:

$$\begin{aligned} 0 &= \nabla_\theta H - \nabla_\theta F - \nabla_H F \nabla_\theta H - \nabla_\xi F \nabla_\theta \xi \\ (I - \nabla_H F)\nabla_\theta H &= \nabla_\theta F + \nabla_\xi F \nabla_\theta \xi \\ \nabla_\theta H &= (I - \nabla_H F)^{-1} (\nabla_\theta F + \nabla_\xi F \nabla_\theta \xi) \end{aligned}$$

The entire gradient is then

$$\nabla_\theta L = \nabla_\theta \ell + \nabla_H \ell (I - \nabla_H F)^{-1} (\nabla_\theta F + \nabla_\xi F \nabla_\theta \xi)$$

Once again, the inverse of $I - \nabla_H F$ appears, and we can approximate it via the polynomial $P(x)$ above.

4.3.1 Computing the gradient

The multiplication are best performed left-to-right; this way, they are vector-matrix multiplies, and not matrix-matrix multiplications. The terms in the gradient calculation have the dimensions below:

$\nabla_H \ell$	$1 \times \lvert H \rvert$
$P(\nabla_H F)$	$\lvert H \rvert \times \lvert H \rvert$
$\nabla_\theta F + \nabla_\xi F \nabla_\theta \xi$	$\lvert H \rvert \times \theta$

Currently, training examples are fed into the system one at a time, and the costs and gradients are accumulated into a batch before updating. We make use of Theano's (Bergstra et al., 2011) tensor library extensively, especially the convenient `Rop, Lop` operators that right or left-multiply the jacobian of a term with a vector.

4.4 Transition Functions

The simplest RNN transition function would be the traditional RNN layer, with an extra term for the next hidden state:

$$h_i = \sigma(W_p h_{i-1} + W_n h_{i+1} + b_i)$$

Preliminary experiments did not show good convergence properties. Alternatively, we modify the Gated Recurrent Unit Transition Function, to be bidirectional. Recall the original GRU equations:

$$\text{GRU} \begin{cases} \text{final hidden} & h_t = (1 - z_t)h_{t-1} + z_t \tilde{h}_t \\ \text{candidate hidden} & \tilde{h}_t = \tanh(Wx_t + U(r_t h_{t-1}) + \tilde{b}) \\ \text{update weight} & z_t = \sigma(W_z x_t + U_z h_{t-1} + b_z) \\ \text{reset gate} & r_t = \sigma(W_r x_t + U_r h_{t-1} + b_r) \end{cases}$$

Now, we replace the h_{t-1} term with $h_{t,c}$, a combination of previous and next hidden states, via a switch:

$$\text{new} \begin{cases} \text{redefine "previous"} & h_{t,c} = s h_{t-1} + (1 - s)h_{t+1} \\ \text{switch} & s = \frac{s_l}{s_l + s_r} \\ \text{left switch} & s_l = \sigma(W_{xsl} x_t + W_{sl} h_{t-1} + b_{sl}) \\ \text{right switch} & s_r = \sigma(W_{xrl} x_t + W_{sr} h_{t+1} + b_{sr}) \end{cases}$$

5 Experiments: Part-of-speech tagging

Part of speech tagging is a natural choice for our 'word classifier' described above. Since part-of-speech tagging is a mature field, our aim is not to build the best tagger in the world, but to show the new architecture in action. To train a part-of-speech tagger, we simply let L be a softmax layer transforming each hidden unit output into a part of speech tag. Initially, ξ consisted only of word vectors for 39,000 case-sensitive vocabulary words. Next, we lowercased the vocabulary words, but added a single feature indicating whether case appeared in the data. Third, we added six additional 'word vector' components to encode the top-2000 most common prefixes and suffixes of words, for affix lengths 2 to 4. Finally, we added in other (binary) features to indicate numbers, symbols, punctuation, and more rich case data, as used by (Huang et al., 2015).

We trained the POS tagger on the Wall Street Journal corpus, blocks 0-18, validated on 19-21, and tested on 22-24. We also tested it on the TED treebank (Neubig et al), and compared it to the results of the off-the-shelf Stanford Part-of-Speech tagger. The results are indicated in Table 1. We were able to achieve comparable results, and as Manning notes, performance gains past that point are quite difficult, due to errors/inconsistencies in the dataset, ambiguity, and very difficult linguistics, sometimes with dependencies across sentences (Manning, 2011).

Training was done using stochastic gradient descent, with an initial learning rate of 0.5. The costs and gradients were all done for individual sentences on a GPU, and then aggregated together in a batch of size 50 before updating the parameters of the model. Word vectors were of dimension 200, prefix and suffix vectors were of dimension 20. Hidden unit size was equal to feature input size, so in this case, 321. Training this way is takes about 20 seconds per batch, and without the GPU, it is approximately the same speed, however it uses all 12 CPU cores of the machine running the experiment. Therefore for a multi-gpu machine it makes more sense to use the GPU and run several experiments in parallel.

We also visualized some of the outputs of the "switch" variables for various sentences. The switch is made up of many features, so it does not necessarily always correspond to human judgment, but by taking the average, one can get a sense of the flow of information. In Figure 3, we see a visualization of the switch on a very simple sentence, and in Figure 4 we see it in action over a more complicated sentence. Interestingly, phrasal structures emerge.

6 Experiments: Sentiment analysis

We used the Stanford IMDB movie review corpus (Maas et al., 2011), training and validating on 25,000 movie reviews (22.5K/2.5K split), and testing on a separate 25,000. The corpora are evenly split into half positive, and half negative reviews. We set it up as a binary classification problem (like/dislike), and use the architecture inspired by Theano's deep learning for sentiment analysis tutorial[1]: after the encoding of the hidden states via either a recurrent net (LSTM or similar) or an implicit network, the hidden states are summed together across words (and averaged) before feeding into a final binary logistic regression classifier. The performance of our model and a few baselines are noted in Table 3. As in the theano example, the model was trained with Adadelta (Zeiler, 2012) because this model seems to train very poorly with stochastic gradient descent. Additionally, we used a batch size of 1, which we surprisingly noted yielded faster convergence in this problem. Because of the extremely long sentences in each training example, we used the third order approximation to the matrix inverse, described earlier.

[1] http://deeplearning.net/tutorial/lstm.html

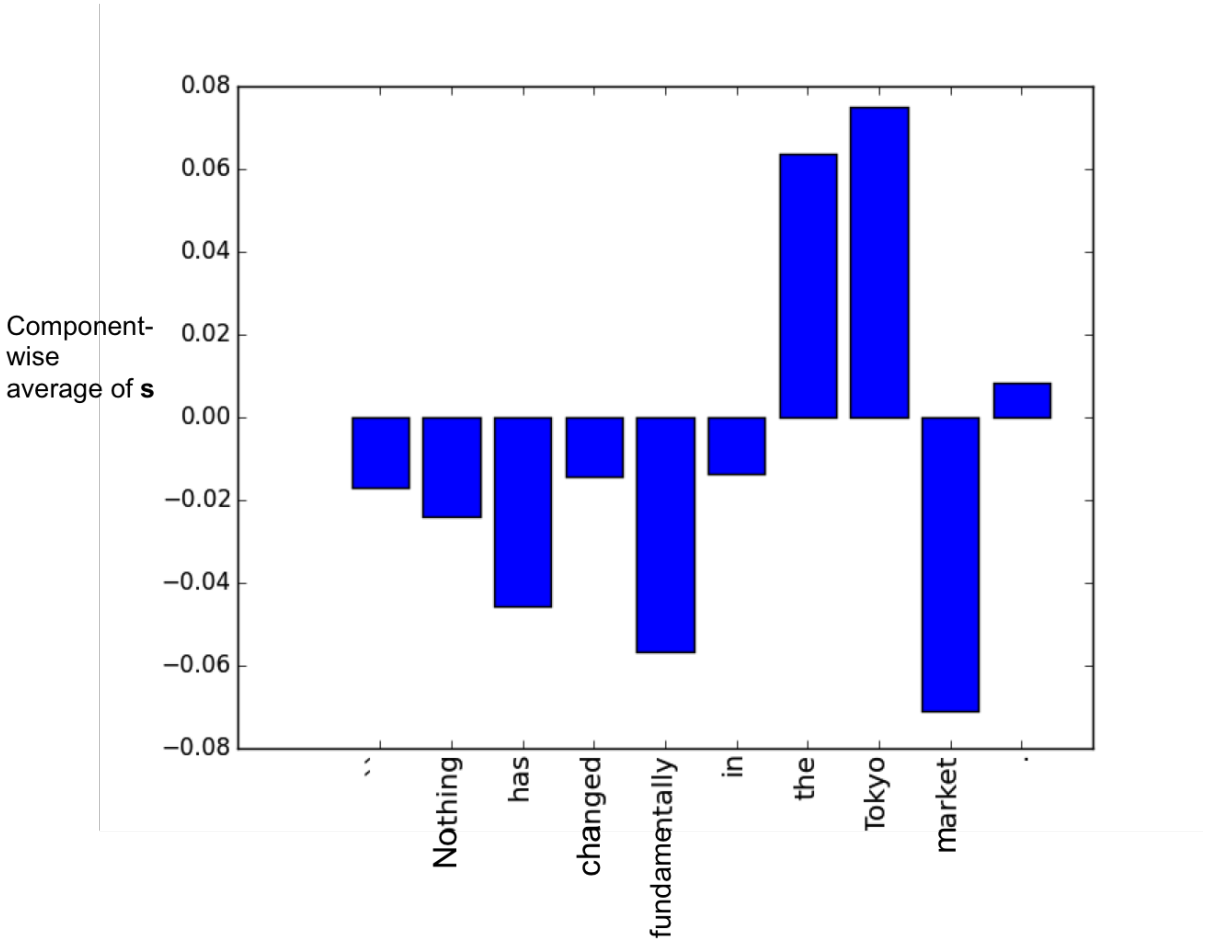

Figure 3: Visualization of the switch variable. Values above 0 indicate a right-to-left flow of information, while values below 0 indicate left-to-right. Note that ‘Tokyo’ is used to modify ‘market’, instead of being a noun, and thus needs information from ‘market’ to make the correct determination.

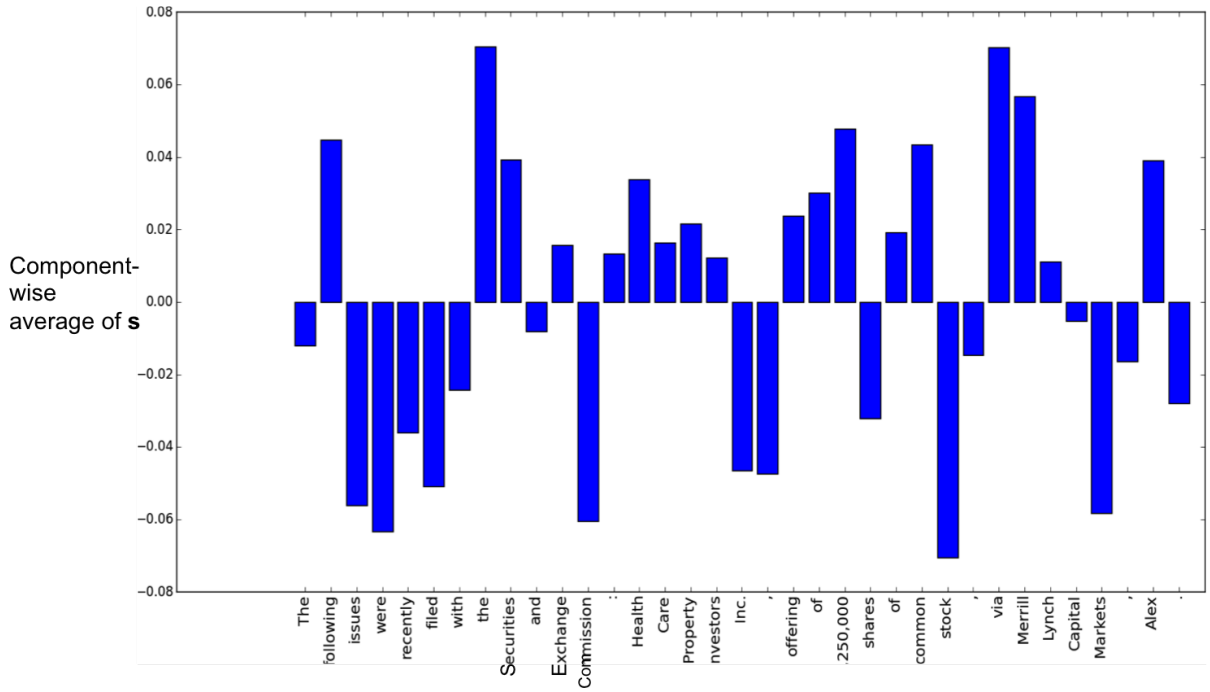

Figure 4: A more common and complicated example. Note the entire clause about 'Health Care Property Investors Inc.' propagates information from the right – without the Inc, it may not realize it is the name of a company. Also of note are phrases "offering of 200,000 shares" and "Merrill Lynch Capital Markets."

7 Acknowledgements

This work would not be possible without the support of the Air Force Research Laboratory, which has always encouraged the pursuit of interesting ideas. Thanks to Brian Thompson for deep learning tips and tricks, Fred Richardson and Nick Malyska for interesting discussion of related work, and Liz Salesky for NLP application suggestions!

Tagger	WSJ Accuracy
Word vectors only	0.9626
Single case feature	0.9650
Ensemble of above (2)	0.9683
Affix word-vectors	0.9714
Ensemble of above (4)	0.9731
Case+Symbol feats	0.9730
Ensemble of above (4)	0.9736
Stanford POS Tagger	0.9732

Table 1: Tagging performance.

Architecture	WSJ Accuracy
Gated recurrent	96.51
LSTM	96.53
Bidirectional GRU	97.26
Bidirectional LSTM	97.27
Implicit	97.30

Table 2: Tagging performance relative to other recurrent architectures.

Architecture	IMDB Accuracy
Gated recurrent	0.8658
LSTM	0.8784
B-LSTM	0.8836
Implicit	0.8810*

Table 3: Sentiment performance relative to other recurrent architectures. *=incomplete result

References

James Bergstra, Frédéric Bastien, Olivier Breuleux, Pascal Lamblin, Razvan Pascanu, Olivier Delalleau, Guillaume Desjardins, David Warde-Farley, Ian Goodfellow, Arnaud Bergeron, et al. 2011. Theano: Deep learning on gpus with python. In *NIPS 2011, BigLearning Workshop, Granada, Spain*.

Kyunghyun Cho, Bart van Merriënboer, Dzmitry Bahdanau, and Yoshua Bengio. 2014. On the properties of neural machine translation: Encoder–decoder approaches. *Syntax, Semantics and Structure in Statistical Translation*, page 103.

Jenny Rose Finkel, Trond Grenager, and Christopher Manning. 2005. Incorporating non-local information into information extraction systems by gibbs sampling. In *Proceedings of the 43rd Annual Meeting on Association for Computational Linguistics*, pages 363–370. Association for Computational Linguistics.

Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural computation*, 9(8):1735–1780.

Zhiheng Huang, Wei Xu, and Kai Yu. 2015. Bidirectional lstm-crf models for sequence tagging. *arXiv preprint arXiv:1508.01991*.

Andrew L. Maas, Raymond E. Daly, Peter T. Pham, Dan Huang, Andrew Y. Ng, and Christopher Potts. 2011. Learning word vectors for sentiment analysis. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies*, pages 142–150, Portland, Oregon, USA, June. Association for Computational Linguistics.

Christopher D Manning. 2011. Part-of-speech tagging from 97% to 100%: is it time for some linguistics? In *International Conference on Intelligent Text Processing and Computational Linguistics*, pages 171–189. Springer.

Mike Schuster and Kuldip K Paliwal. 1997. Bidirectional recurrent neural networks. *Signal Processing, IEEE Transactions on*, 45(11):2673–2681.

Matthew D Zeiler. 2012. Adadelta: an adaptive learning rate method. *arXiv preprint arXiv:1212.5701*.